HP-15C Nth-degree Polynomial Fitting

Valentín Albillo (Ex-PPC #4747, HPCC #1075)

It frequently happens, both in theoretical problems and in real-life applications, that we've got a series of data points (x,y) and we need to fit some smooth curve, defined by a mathematical function, that passes through all of them. Once we've got a suitable function, we can then perform such things as interpolation, inverse interpolation, integration, differentiation, etc. Among the many functions we could fit to the data, polynomials are by far the easiest to evaluate and manipulate, either numerically or symbolically, so our problem is reduced to this: given a set of **n+1** data points (x_0, y_0), …, (x_n, y_n), find an nth-degree polynomial,

$$P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + ... + a_n x^n$$

such that $P(x_i) = y_i$ for i = 0, 1, …, n. This results in the following system of **n+1** *linear* equations to determine the unknown coefficients $a_0, a_1, ..., a_n$:

$$\begin{gathered} a_0 + a_1 x_0 + a_2 x_0^2 + a_3 x_0^3 + ... + a_n x_0^n = y_0 \\ a_0 + a_1 x_1 + a_2 x_1^2 + a_3 x_1^3 + ... + a_n x_1^n = y_1 \\ \dots\dots\dots\dots\dots\dots\dots \\ a_0 + a_1 x_n + a_2 x_n^2 + a_3 x_n^3 + ... + a_n x_n^n = y_n \end{gathered}$$

which we'll construct and then solve using the ***powerful matrix capabilities*** of the HP-15C. The maximum size for **n** is bounded by available memory, which for the 15C amounts to a maximum of 64 registers available to store matrix elements.

For an nth-degree polynomial, the above system requires one (n+1)*(n+1) matrix and a column vector which will hold both the y_i data and then the a_i coefficients. Thus, the memory requirements to fit an nth degree polynomial are such that (n+1)*(n+2) <= 64.

The maximum value of n which satisfies the expression above is 6, so we can fit a 6th-degree polynomial, which requires 7*8 = 56 registers, thus leaving only 8 registers for the program and any auxiliary registers needed. Assuming we'll use at least one extra storage register for loops, etc., this leaves just 7 registers = 49 bytes for the program itself, and as some matricial instructions take more than 1 byte, this means we have *less* than 49 steps for the program. Can it be done ? Yes !

The program

Here is a very small program for the HP-15C that I wrote specifically for this article. It’s exactly 49 bytes long (including an *implied* **RTN** instruction at the very end). It’ll prompt for data input, construct and solve the system, and output the resulting coefficients, in as few as 42 program lines.

Program listing

- **Very Important**: steps 20, 23, 32, 38 and 41 ***must*** be entered ***in*** USER mode, while all the rest ***must*** be entered ***out*** of USER mode. An “**u**”-like character must appear next to the step number *only* for steps 20, 23, 32, 38 and 41, and no others.

```
01   LBL A        15   R/S         29   LBL B
02     1          16   ENTER       30   RCL 0
03     +          17   ENTER       31   R/S
04   STO 2        18   ENTER       32  uSTO B
05   ENTER        19     1         33   GTO B
06   DIM A        20  uSTO A       34   RCL MATRIX B
07     1          21   LBL 1       35   RCL MATRIX A
08   DIM B        22     *         36   RESULT B
09   MATRIX 1     23  uSTO A       37     ÷
10   LBL 0        24   LBL 2       38  uRCL B
11   RCL DIM B    25   DSE I       39   LBL 3
12     -          26   GTO 1       40   R/S
13   STO I        27   DSE 2       41  uRCL B
14   RCL 0        28   GTO 0       42   GTO 3
```

Data Register usage

RI	loop columns
R0	row index for matrix elements
R1	column index for matrix elements
R2	loop rows

Notes

- steps 01-09 take the degree in X and dimension both matrices depending on it.

- steps 10-28 fill up the matrix row by row, setting up an internal loop (steps 21-26) to compute and store all required powers of x_i . Notice how the single *user* **STO** at step 23 does most of the work.
- once all x_i have been input, steps 29-33 are a very tight loop to prompt for the y_i and store them in the vector. The *user* **STO** at step 32 does *triple duty* by storing each value in its proper place in the vector, updating the indexes, and testing for loop termination.
- when all data points have been entered, steps 34-37 solve the system of equations, with the simple-looking "division" operation at step 37 actually doing the equivalent of a matrix inversion and subsequent matrix product with 13-digit internal precision at microcode speeds (21 sec. for a 7x7 system). The matrix is left in *LU-decomposed form*, so allowing for the fast computation of the coefficients of another polynomial with the *same* x_i but a *different* set of y_i .
- finally, the loop at steps 39-42 (actually, a RCL twin of the STO one at steps 29-33) outputs the computed coefficients, one at a time.

Usage instructions

1) After keying in the program, make sure you're *not* in complex mode (press **CF 8** if in doubt), then you must commit enough storage registers to the common pool for the matrix operations. To that effect, press:

   N, **f DIM (i)**

   where N depends on the degree of the polynomial, as seen in this table:

Degree	N, f DIM (i)	MEM	DIM A	DIM B
1	2<= N <=52	52 6 7-0	2x2	2x1
2	2<= N <=46	46 12 7-0	3x3	3x1
3	2<= N <= 38	38 20 7-0	4x4	4x1
4	2<= N <=28	28 30 7-0	5x5	5x1
5	2<= N <=16	16 42 7-0	6x6	6x1
6	N = 2 (exact)	2 56 7-0	7x7	7x1

Note: if you get an error, this means you have registers commited to other matrices: redimension the ones you don't need to 0x0 or even easier, redimension *all* matrices to 0x0 executing **MATRIX 0**.

For instance, if you're going to fit a *3rd-degree* polynomial to a set of 4 data points, you may use **38, f DIM (i)**, but any number from 2 to 38 will do as well. On the other hand, if you're going to fit a *6th-degree* polynomial, you ***must*** use

2, f DIM (i). If you don't plan to use any other matrices or registers, you might simply execute **2, f DIM (i)** in all cases.

2) Now, enter the degree of the polynomial you want to fit (1 to 6), and start the program by pressing either:

f PRGM, R/S or **GSB A** or (*in User mode*) **A** → 1.0000

the program will prompt you to enter the x value of the 1st data point. (you must enter *all the x values first, then the y values*):

x_1, R/S → 2.0000
x_2, R/S → 3.0000
...
x_{n+1}, R/S → 1.0000

3) Now the program is prompting you to enter the y value of the 1st data point:

y_1, R/S → 2.0000
y_2, R/S → 3.0000
...
y_{n+1}, R/S → a_0, R/S → a_1, ... , R/S → a_n

and the polynomial which fits all ***n+1*** data points is:

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + ... + a_n x^n$$

4) If you want to display all the coefficients again, set ***User*** mode and press:

f MATRIX 1
RCL B → a_0, RCL B → a_1, ... , RCL B → a_n

4) If you want to compute the coefficients for *another* set of data with the *same* x_i values but *different* y_i values, you don't need to reenter the x values. Press:

GSB B or (*in User mode*) **B** → 1.0000

and go to (3) above to enter the new y_i values. You'll notice that it takes *much less time* to compute the coefficients now, as the matrix is already *in LU-decomposed* form, and thus can be used as is, saving a lot of processing time.

Example 1: Data fitting and predictions

A series of measurements has resulted in the following table of values corresponding to one independent variable x, and two dependent variables y, z. As there are 7 triplets in all, fit a couple of 6th-degree polynomials y=y(x) and z=z(x) to the data and use them to predict the values of y(x) and z(x) for x =-1.0, 0.0, +1.0

Var.	***1st***	***2nd***	***3rd***	***4th***	***5th***	***6th***	***7th***
X	*-1.3118*	*-0.5663*	*-0.1503*	*+0.0323*	*+0.6236*	*+0.9312*	*+1.4202*
Y	*-0.9666*	*-0.5365*	*-0.1497*	*+0.0323*	*+0.5840*	*+0.8023*	*+0.9887*
Z	*+0.2561*	*+0.8439*	*+0.9887*	*+0.9995*	*+0.8118*	*+0.5969*	*+0.1500*

As you can see, *the x values aren't equally spaced*, and although in this particular example they're input in order of increasing value, that's *not* necessary either. So, we're going to compute the a_i,b_i coefficients for these two 6th-degree polynomials:

$$y(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + ... + a_6 x^6$$
$$z(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3 + ... + b_6 x^6$$

First, we need to commit enough registers and clear all matrices by executing:

2, f DIM (i), f MATRIX 0

Now, input the degree (6), start the program, and at the prompts, enter all data points for the 1st polynomial, first the x values, then the y values. In User mode, and FIX 5:

6, A → 1.00000 [*prompt to enter* x_1]
-1.3118, R/S → 2.00000 [*prompt to enter* x_2]
-0.5663, R/S → 3.00000 [*prompt to enter* x_3]
... ...
1.4202, R/S → 1.00000 [*prompt to enter* y_1]
-0.9666, R/S → 2.00000 [*prompt to enter* y_2]
... ...
0.9887, R/S → 0.00001 [a_0 *coefficient (after just 25 seconds)*]
R/S → 0.99985 [a_1 *coefficient*]
... ...
R/S → 0.00027 [a_6 *coefficient*]

so we've got our first polynomial, y(x):

$$\underline{y = 0.00001+0.99985 x+0.00021 x^2-0.16592 x^3-0.00062 x^4+0.00759 x^5+0.00027 x^6}$$

Interpolated values are y(-1.0) = <u>-0.84164</u>, y(0.0) = <u>0.00001</u>, and y(1.0) = <u>0.84139</u>

To fit the z values, as the x values are the same, *we don't need to reenter them*, we just need to input the z values. Assuming we're still in User mode and FIX 5:

```
              B         →  1.00000   [prompt to enter z₁]
   0.2561,  R/S      →  2.00000   [prompt to enter z₂]
         ... ... ... ... ... ... ... ... ... ...
   0.1500,  R/S      →  1.00001   [b₀ coefficient (after only 9 seconds !)]
            R/S      →  0.00026   [b₁ coefficient]
         ... ... ... ... ... ... ... ... ... ...
            R/S      →  -0.00174  [b₆ coefficient]
```

and thus we've got our second polynomial, z(x) (*only much faster !*):

<u>$z = 1.00001+0.00026\,x-0.50021\,x^2-0.00089x^3+0.04248\,x^4+0.00044\,x^5-0.00174\,x^6$</u>

Interpolated values are z(-1.0) = <u>0.54073</u>, z(0.0) = <u>1.00001</u>, and z(1.0) = <u>0.54035</u>

The original data points were taken from **y=sin(x)** and **z=cos(x)**, and now we can check that our interpolated results are indeed correct to 4 decimal places as well.

Example 2: Constructing a polynomial knowing its roots

Find a 6th-degree polynomial P(x) such that its roots are 1/3, 1, sqrt(5), Pi, e, sqrt(10) and P(Ln 2) = sin(1) ("sqrt" is the square root function)

In **User** mode, **Rad** mode, and FIX 5, the keystroke sequence:

6, A,
3, 1/X, R/S, 1, R/S, 5, Sqrt, R/S, Pi, R/S, 1, e^x, R/S, 10, Sqrt, R/S,
2, Ln, R/S, 0, R/S, 0, R/S, 0, R/S, 0, R/S, 0, R/S, 0, R/S, 1, Sin, R/S

gives:

$$P(x) = \underline{-8.12096 + 44.25624\,x - 77.80622\,x^2 + 62.24622\,x^3 - 25.25201\,x^4 + 5.08018\,x^5 - 0.40346\,x^6}$$

Example 3: Inverse interpolation

Find a 5th-degree polynomial x(k) which will approximate the root of $\boldsymbol{x^3 + x - k = 0}$ *for values of k between 2 and 10. Predict the value of the root x when k = 5.*

First we evaluate $\boldsymbol{k = x^3+x}$ for x = 1.0, 1.2, 1.4, 1.6, 1.8 and 2.0, to obtain these six data points (k, x):

(2.0, 1.0), (2.928, 1.2), (4.144, 1.4), (5.696, 1.6), (7.632, 1.8) and (10.0, 2.0).

Notice we've *reversed* the order of x and k, because we're *not* interested in the value of k given x, but *in the value of x given k.* Now, proceed as above to find the 5th-degree polynomial which fits those data points. In FIX 5, our program gives:

$$\underline{x(k) = 0.28469 + 0.50634\,k - 0.09666\,k^2 + 0.01292\,k^3 - 0.00094\,k^4 + 0.00003\,k^5}$$

which, for k=5 gives $\underline{\boldsymbol{x = 1.5158}}$. As the real positive root of $\boldsymbol{x^3 + x - 5 = 0}$ comes out as x = 1.5160, our polynomial actually gave nearly *five* significant digits correct.

Final remarks

It's amazing how the advanced capabilities and programming features of the HP-15C allow it to fast and easily solve, in just a few program steps, problems that would take *hundreds* of program lines and *much longer* execution times in all other programmable calculators of its time. Even today, despite the tremendous advances in technology, it still demonstrates what you can do with a machine so small it fits any pocket size and which keeps on running long after the Duracell bunny has rusted away !

Note: For an HP-41C/Advantage implementation, see my article “Long Live the HP-41C Advantage !” elsewhere in this issue. There you’ll find a thorough discussion on the ability of the Advantage ROM to make available the advanced matrix capabilities of the HP-15C in the HP-41C realm.